**Reporte**

**Box Blur Algorithm Parallelization**

*Julian Esteban Santos Martínez, Johan Sebastián Sanchez Vargas, Cesar Augusto Solano Corzo*

*Universidad Nacional de Colombia, Facultad de Ingeniería, Bogotá, Colombia*

1. **Introducción**

En este documento se presenta la aplicación de la paralelización de un programa cuyo objetivo es aplicar un efecto de difuminado a imágenes de distintas resoluciones. Se paralelizará con diferentes herramientas (OpenMP, CUDA y MPI) y se hará una comparación de rendimiento entre estas aplicaciones y su algoritmo secuencial con el fin de determinar qué método tiene mejor eficiencia y concluir que aplicación del programa es mejor.

1. **Funcionamiento**

**Secuencial**

El programa consta de 2 procesos principales. En primer lugar, maneja la lectura y escritura de imágenes y en segundo lugar, realiza el difuminado de la imagen proveída.

Para manejar la lectura y escritura de imágenes el programa utiliza varias estructuras para definir las propiedades de una imagen, incluye los colores RGB, los valores del ancho y alto de la imagen y los bytes con los que funciona una imagen con formato bmp. Para identificar una imagen el programa lee las propiedades del archivo y les asigna memoria de manera dinámica con la función malloc. El resultado de esta operación se almacena en una estructura de tipo imagen a la que se le modifican sus valores de RGB para cada píxel, esta modificación permite definir cualquier tipo de efecto visual sobre la imagen, en este caso el programa efectúa solamente el efecto de difuminado

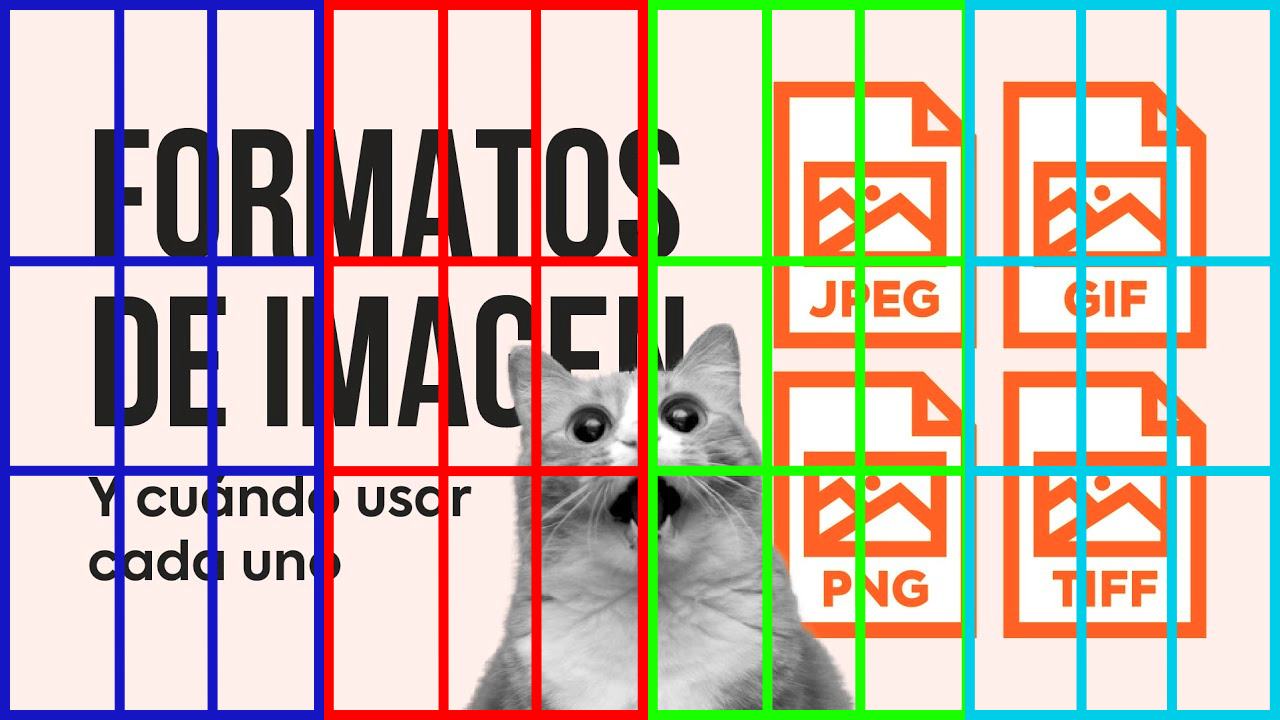
Para difuminar la imagen el programa utiliza un kernel como base, este Kernel es una matriz cuadrada cuyo tamaño es definido por el usuario. El programa evalúa secuencialmente partes de la imagen del mismo tamaño que el kernel y asigna en el píxel central la sumatoria de todos los píxeles del sector evaluado, luego los divide por el tamaño del kernel. El resultado de esta operación es que los valores RGB de cada píxel en la imagen se asemejan más a los valores RGB de los píxeles vecinos.

En la siguiente imagen se muestra como el program evalua la imagen mediante matrices generadas en cada pixel, en este ejemplo se utiliza un kernel de 3, por lo que se genera una matriz de 3x3 y en el punto central se asigna la el promedio de los pixeles aledaños.



**OpenMP**

Al utilizar la librería OpenMP se implementó el modelo de Block Cyclic Data para el manejo de la información. Este modelo reparte equitativamente las tareas a los hilos del procesador mediante unidades denominadas *chunks.* Para procesar la imagen, el primer paso es leer la información RGB de los píxeles, a continuación se almacena en una matriz del tamaño de filas y columnas de la resolución de la imagen. El modelo block cyclic data entonces se encarga de repartir el procesamiento de la matriz equitativamente entre la cantidad de hilos designados. Fila por fila se divide la matriz en la cantidad de hilos establecida y se le da una parte o un *chunk* a cada hilo en cada iteración. En resumen, lo que se hace es dividir la matriz por columnas, la cantidad de columnas es igual a la cantidad de hilos, y entregarle una columna a cada hilo para su procesamiento.



En la imagen anterior se muestra como sería la distribución para cuatro hilos, la imagen se divide en cuatro columnas y en cada iteración la fila se divide y se reparte equitativamente. Al finalizar el programa cada hilo se habrá encargado de una columna representada por los colores.

1. **Experimentos**

Para probar el algoritmo creado para aplicar un efecto borroso en las imágenes, se pasaron como entrada diferentes imágenes con distintas resoluciones, obteniendo los resultados mostrados a continuación:

720p:



1080p:



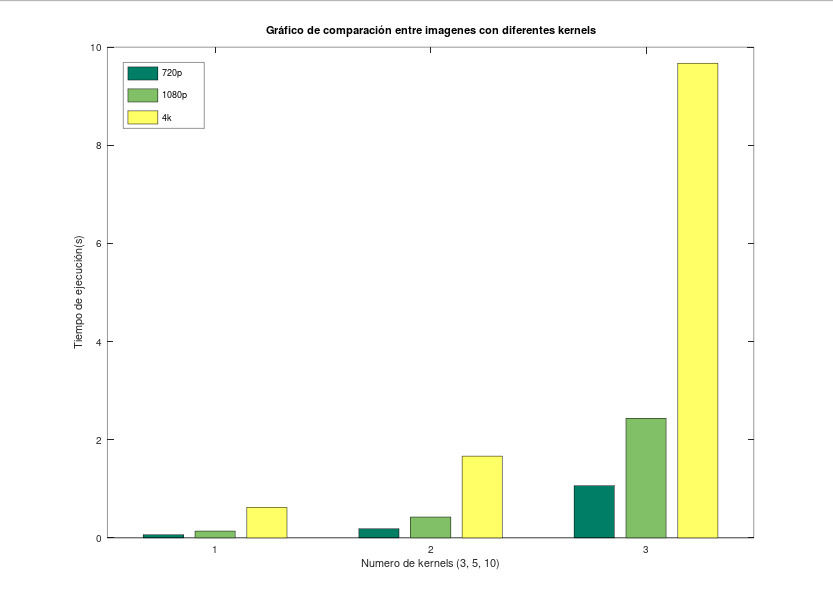


4k:





1. **Resultados**

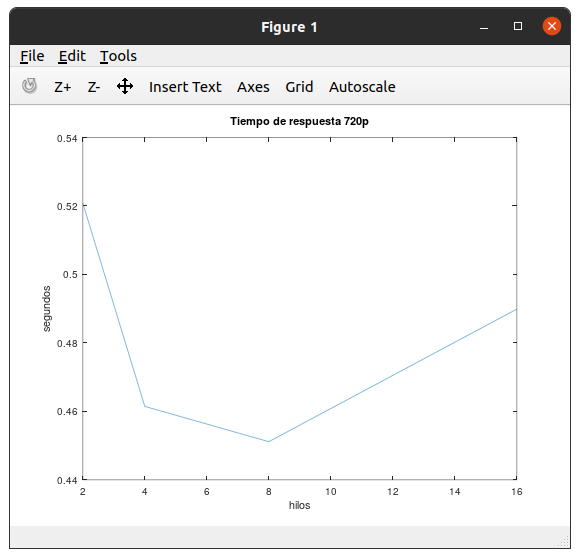
****

Para el algoritmo secuencial realizamos 50 veces cada filtrado modificando el tamaño del Kernel usado, en la gráfica anterior podemos observar la comparación de las tres resoluciones, 720p, 1080p y 4k, aplicadas a tres diferentes tamaños de kernels, matriz de 3x3, 5x5 y 10x10. Por ejemplo, el promedio de la imagen de resolución 720p aplicada a un kernel de 3x3 nos dio un tiempo de 0.065032. Este valor comparado a el tiempo de respuesta de la imagen 4k aplicada al mismo kernel que su tiempo es de 0.617208 es bastante considerable. De la misma forma con los demás tamaños de kernel podemos observar un crecimiento bastante exponencial en nuestro programa secuencial.

Por otro lado, al realizar la paralelización del programa con la API OpenMP se probó el algoritmo con distinta cantidad de hilos en cada una de las imágenes. Se hicieron pruebas por cada imagen con cada cantidad de hilos predeterminada: 2, 4, 8 y 16. Se tomaron los tiempos de respuesta y los speed-ups de cada ejecución del programa, y luego se calculó el promedio de todos los datos obtenidos. Estos promedios se pueden ver graficados y tabulados en las siguientes tablas y figuras:

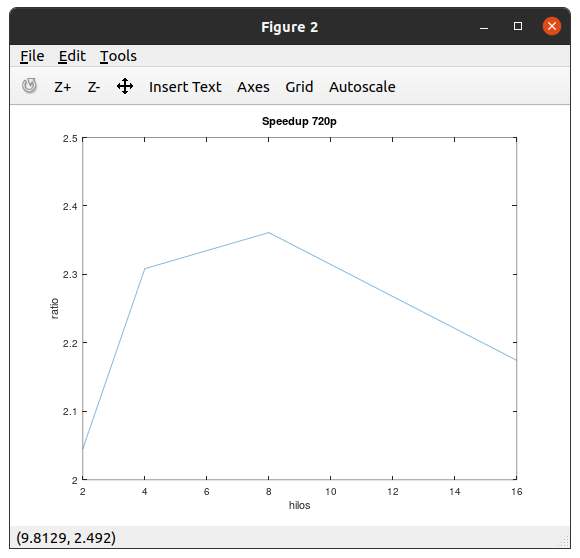
| Hilos | Tiempo de respuesta |
| --- | --- |
| 2 | 0.52081 |
| 4 | 0.46144 |
| 8 | 0.45114 |
| 16 | 0.48986 |

*Tabla 1. Tiempos de respuesta con una imagen de resolución 720p*



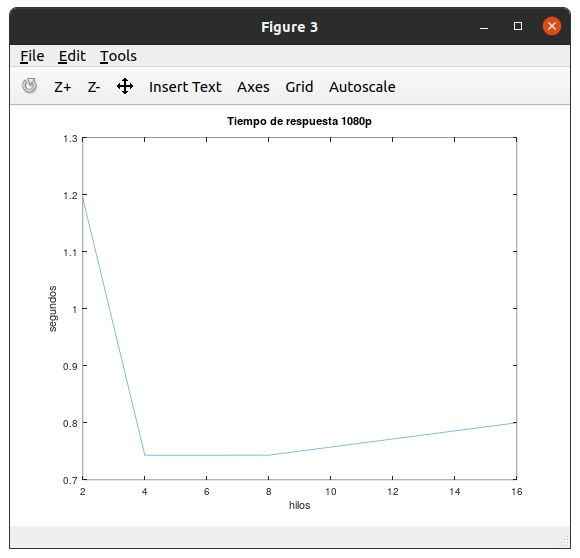
| Hilos | Speed-up |
| --- | --- |
| 2 | 2.0452 |
| 4 | 2.3084 |
| 8 | 2.3611 |
| 16 | 2.1745 |

*Tabla 2. Speed up de imagen con resolución 720p*



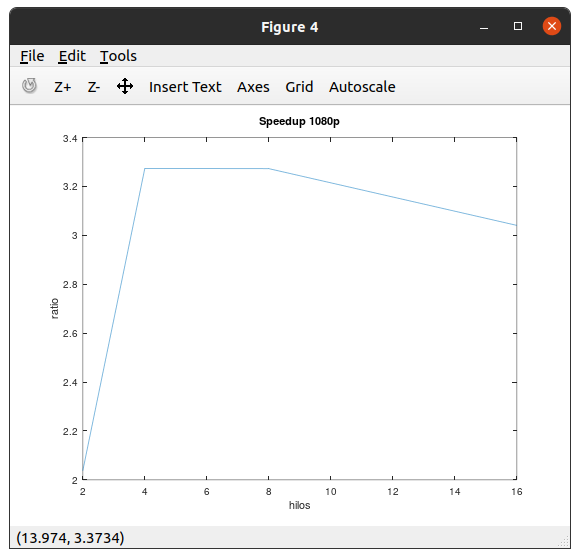
| Hilos | Tiempo de respuesta |
| --- | --- |
| 2 | 1.1938 |
| 4 | 0.74303 |
| 8 | 0.7432 |
| 16 | 0.79992 |

*Tabla 3. Tiempos de respuesta con una imagen de resolución 1080p*



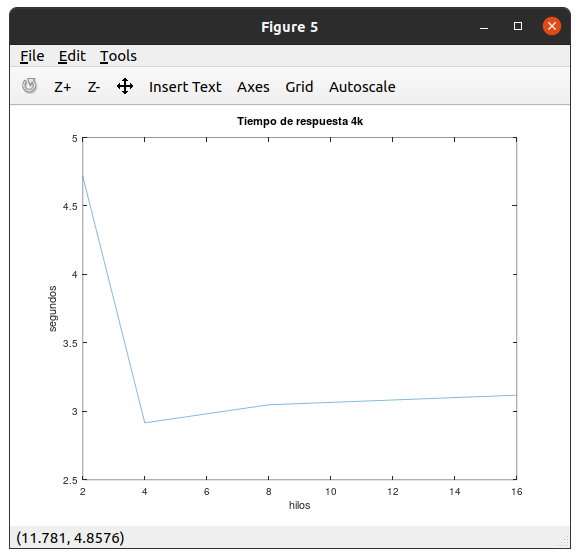
| Hilos | Speed-up |
| --- | --- |
| 2 | 4.9745 |
| 4 | 5.6145 |
| 8 | 5.7428 |
| 16 | 5.2888 |

*Tabla 2. Speed up de imagen con resolución 1080p*



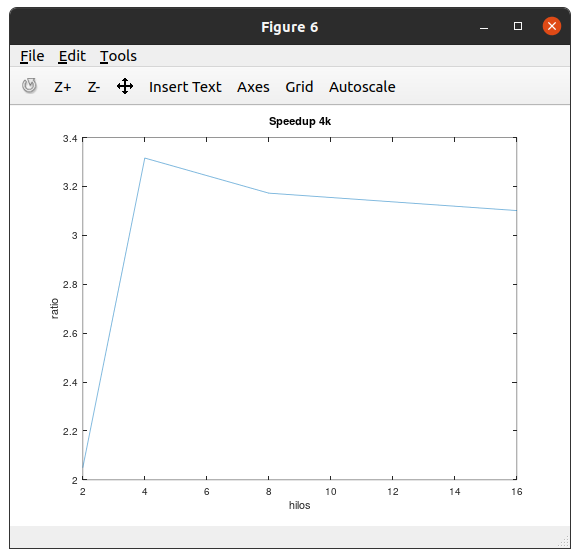
| Hilos | Tiempo de respuesta |
| --- | --- |
| 2 | 4.7165 |
| 4 | 2.9159 |
| 8 | 3.0478 |
| 16 | 3.1179 |

*Tabla 5. Tiempos de respuesta con una imagen de resolución 4k*



| Hilos | Speed-up |
| --- | --- |
| 2 | 2.05 |
| 4 | 3.316 |
| 8 | 3.1725 |
| 16 | 3.1012 |

*Tabla 6. Speed up de imagen con resolución 4k*



En las gráficas y datos podemos ver que en todas las resoluciones se sigue un patrón, donde agregando más hilos a la paralelización, se van reduciendo los tiempos de respuesta. Es decir, se tiene un mejor rendimiento correr el programa paralelizado que de manera secuencial.

Sin embargo, al probar y correr el código en una máquina de 4 núcleos físicos y 4 virtuales, no se ve una mejora cuando se corre el programa con 16 hilos, incluso se ve un empeoramiento tanto en speedup como en tiempos de respuesta teniendo el pico de rendimiento en los 8 hilos. Esto se debe a que al ser una gran cantidad de hilos que el computador prueba no puede manejar, se genera un cuello de botella, y termina demorando más el proceso de aplicar el filtro a las imágenes.

1. **Conclusiones**

**Secuencial**

Se debe encontrar un punto de equilibrio entre el kernel utilizado y el rendimiento del programa. A partir de un kernel de 64 la imagen está difuminada en gran medida pero si se aumenta el valor el cambio no se aprecia mucho y se genera una carga de procesamiento mucho mayor en el programa.

El programa nos ayudó a comprender el funcionamiento de los archivos bmp debido a los headers y los bytes con los que se manejan con lo que se pudo establecer un mejor acercamiento al difuminado buscado.

**OpenMP**

Como era de esperarse el modelo paralelizado resultó ser mucho más eficiente para procesar la imagen, fue especialmente más eficiente en imágenes de mayor resolución.

Para el procesador utilizado de ocho hilos se ve que el punto de mejor rendimiento es la totalidad de los hilos, valores superiores reducen el rendimiento del programa.

En las imágenes de resolución 4k se ve una mejoría de hasta casi un 500% en la velocidad de procesamiento.